

1. Stunde

Klassen und Objekte:

Beispiel: Klasse Auto

Die Klasse Auto besitzt Attribute und Methoden

Attribute: Farbe, Größe, Kraftstoffart, Masse, Baujahr

Methode: Vorwärts, Rückwärts, Lenken, Bremsen, Neu, Vernichten

Eine Instanz der Klasse Auto sei folgendes Objekt

Java

MeinAuto = Neu Auto

MeinAuto

XBase

MeinAuto := Auto:neu() // Attribute können durch := verändert werden

MeinAuto:Farbe := „rot“ // rot ist der Attributwert

MeinAuto:Baujahr := 2011 ...

MeinAuto:Erzeugen()

MeinAuto:Vorwärts(50) // Methode wird mit () aufgerufen

2. Stunde

Einrichten der Programmierumgebung

1. Quelltext erfassen (kein Textendezeichen, keine Unterscheidung der Groß- und Kleinschreibung)
2. Kompilieren. Dabei wird der Quelltext in sogenannten Objektcode umgesetzt und der Quelltext auf Fehler überprüft.
3. Linken. Der eigene Objektcode wird mit vorgegebenem Objektcode der Programmiersprache zu einem eigenständigen Programm zusammengeführt.
=>EXE-Datei (Anwendung)

Schritt 2 und 3 werden durch eine Projekt-Datei zusammengeführt. Dazu wird eine Datei benötigt, die fest vorgegeben ist.

Project.xpj => muss in jedem Verzeichnis sein, in dem wir programmieren

Die Datei autoxpp.bat setzt die Programmierumgebung. Diese sollte sich unter Z:\ befinden.

1. Start – Ausführen - „CMD“ - <Enter>
2. Den Befehl „autoxpp“ in dem DOS-Fenster eingeben
 - cd prog (cd change directory, wechselt das Verzeichnis)
 - dir (Ausgabe des Inhaltes eines Ordners/Verzeichnis)
 - md (make directory, Verzeichnis erstellen)
3. In das passende Verzeichnis wechseln mit cd ...
4. Quelltext erfassen

5. `pbuild project` zum kompilieren und linken
Unser erstes Programm:

3. Stunde

Formdesigner aufrufen mit „`xppfd`“

Besprechung des Programms.

4. Stunde

Aufgabe: Der 2. Button soll nach dem Anwählen verschwinden.

Lsg.: In Activate den Befehl `oXbp:hide()` einfügen

- Nach Änderung des Quelltextes muss gespeichert werden
- Jetzt muss IMMER neu kompiliert werden (`pbuild...`)
Hilfe: Pfeiltasten für die alten Befehle
- PROGRAMM.EXE created s...
- programm

Aufgabe: Der 1. Button soll den zweiten Button wieder anzeigen lassen.

Lsg.: Es müssen Variablen eingeführt werden, die die Objekte speichern.

Bedeutung der drawing area

Jedes Objekt muss einem Parent zugeordnet werden. Der Parent wird auch als Eigentümer bezeichnet.

Das Hauptfenster „gehört“ Windows. Die weiteren Button werden im Programmfenster verankert, in dem sie als Parent die Zeichenebene vom Fenster erhalten.

Bsp:

```
xbpObjekt:new( odlg:drawingarea, ....)
```

Aufgabe:

Erstelle zwei Fenster und ordne je einem Fenster einen Button zu.

Lsg: `oWin1` bzw. `oWin2` als Parent zuordnen.

Aufgabe:

Erstelle zwei Fenster. Das erste beinhaltet zwei Button, „Erstellen“ und „Vernichten“, das zweite Fenster beinhaltet drei Buttons, die beim Anklicken von Erstellen alle erscheinen und beim Anklicken von Vernichten alle verschwinden. Die drei Buttons vom zweiten Fenster verschwinden ebenfalls, wenn sie einzeln angeklickt werden.

4. Stunde

Aufgabe: Erstelle ein Programm, das ein Eingabefeld, ein MLE-Feld und zwei Button enthält. Nach Anklicken des ersten Buttons soll der Inhalt des Eingabefeldes in einer Confirmbox angezeigt werden, beim Anklicken des zweiten der Inhalt des MLE-Feldes.

Lsg: Zunächst müssen wieder die Objekte in Variablen abgespeichert werden. Dann kann der Inhalt der Felder über die gemeinsame Methode :editBuffer() abgefragt werden.

Bsp.: ConfirmBox(oDlg:drawingArea ,oMLE:editBuffer(), ...

5. Stunde

Aufgabe: Der Text des SLE-Feldes soll nach Betätigung eines Buttons in das MLE-Feld geladen werden.

Lsg.: oMLE:setdata(oSLE:editbuffer())

Die nächsten Objekte: Radiobutton, Checkbox

Aufgabe: Erstelle ein Fenster mit 2 Radiobutton, zwei Checkboxes und einem Abbruch-Button. Beim Klick auf den Button soll der Zustand der Radiobutton abgefragt werden.

Lsg...

Aufgabe:

Erstelle ein Programm, das eine Listbox, ein SLE-Feld und zwei Buttons enthält. Beim Klick auf den ersten Button soll der Inhalt des SLE-Feldes in die Listbox übertragen werden.

Lsg:

Im Pushbutton muss folgender Quellcode eingetragen werden

```
oXbp:activate := {|| oList:additem(oSLE:getdata()) }
```

Erweiterung:

Mit Hilfe des zweiten Pushbuttons soll die Listbox wieder komplett geleert werden.

Lsg:

Folgender Quelltext ist zu ergänzen

```
oXbp:activate := {|| oList:clear() }
```

Aufgabe 2 zu Listboxen:

Ein Programm soll zwei Listboxen und ein SLE Feld enthalten. Der Inhalt des SLE-Feldes soll in die 1. Listbox übertragen werden.

- Nach Doppelklick auf den Eintrag wandert dieser in die 2. Listbox (u.u)
- Das SLE-Feld soll nach der Eingabe gelöscht werden
- Die beiden Listboxen sollen durch den zweiten Button gelöscht werden

Lösung zu 3:

```
oXbp:activate := {|| Gather(aEditControls), ADRESSEN->(dbappend()),
SCATTER(aEditControls) }
```

Lösung zu 6:

```
oXbp:activate := {|| Gather(aEditControls), ADRESSEN->(dbdelete()),
SCATTER(aEditControls) }
```

So ist der Datensatz immer noch sichtbar.

Erklärung: Der Datensatz hat nur eine Löschmarkierung erhalten. Möchte man diese Datensätze unterdrücken, so schaltet man dies durch den Schalter „set delete on“ ein. Möchte man alle löschmarkierten Datensätze endgültig löschen, so muss man zusätzlich den Befehl „dbpack()“ eingeben. Der Befehl dbzap() löscht sofort ALLE Datensätze (ACHTUNG !!!).

Nach dem Setzen der Löschmarkierung ist der Datensatz immer noch am Bildschirm zu sehen. Deswegen sollte man direkt nach dem Löschen einen Datensatz vorrücken und den nächsten Datensatz anzeigen, damit der Anwender die Veränderung sieht.

zu 4) Das Ende der Datenbank kann durch die Funktion eof() (end of file) abgefragt werden. Der Anfang der Datenbank durch bof() (beginning of file).

Diese Funktionalität bauen wir in eine Funktion ein.

Es gibt zwei verschiedene Typen. Prozeduren und Funktionen. Funktionen können ein Ergebnis zurückgeben, Prozeduren nicht. Da aber Funktionen auch eine leere Rückgabe haben können, finden Prozeduren keine Verwendung mehr. Aufbau:

```
function Name_Der_Funktion()
    [Quelltext]
return nil
```

In einer Quelltextdatei können bel. viele Funktionen angegeben werden.

Benötigt wird eine IF-Abfrage:

```
if IBedingung
    [Quelltext]
else
    [Quelltext]
endif
```

Bsp: if x=8

```
    x := x - 1
else
    x := x + 1
endif
```

Aufgabe: Ergänze eine Listbox.

In einer neuen Funktion soll die Listbox mit den Nachnamen gefüllt werden. Die neue Funktion soll „Liste_fuellen“ heißen.

Der Aufruf muss folgendermaßen erfolgen: Liste_fuellen(oList)

```

Function Liste_fuellen(oList)
adressen->(dbgotop())
oList:clear()
do while !eof()
    oList:addItem(adressen->name)
    adressen->(dbskip())
enddo
return nil

```

Selectsort

Funktionsweise

- Sortiert von der ersten Position ausgehend
- tauscht pro Durchgang einmal
- nimmt im ersten Durchgang das erste Element als Vergleichselement und sucht in der Liste nach einem kleineren Element (merkt sich den Wert und die Position)
- am Ende der Liste ist der Wert der kleinsten Zahl und die Position bekannt
- jetzt erfolgt der Tausch mit der ersten Position

Vorteil zu Bubblesort:

Im ersten Durchgang $n-1$ Vergleiche (entspr. Bubblesort) aber nur eine Vertauschung

Insgesamt ergeben sich bei Selectsort für den BC ein Laufzeitverhalten bei den Vertauschungen von 0 und bei den Vergleichen von n^2 . Beim WC liegt der Wert bei den Vergleichen wieder bei n^2 , bei den Vertauschungen nur bei n . Somit müsste der Selectsort schneller sein als der Bubblesort.

Untersuchung der Laufzeit.

Ergänze die beiden Sortierverfahren um eine Zeitmessung. Dafür ex. die Funktion „seconds()“, die die Sekunden nach Mitternacht angibt (bis auf 1/100 s).

Vor der „Do While“ Schleife merkt man sich die Zeit durch `nSek:=seconds()`. Direkt nach der Schleife kann man nun die Zeitdifferenz in einer Confirmbox ausgeben. Dazu muss der numerische Wert durch die Funktion `str()` umgewandelt werden.

`str(nZahl, nLänge, nDec)`: Als erster Parameter wird die Variable übergeben, dann die gesamte Länge incl. Punkt und dann die Nachkommastellen.

Bsp.: `str(seconds()-nSek, 8, 2)`

Ergebnis:

Der Selectsort benötigt erwartungsgemäß weniger Zeit um eine bestimmte Datenmenge zu sortieren. Allerdings kann man die Linearität beim Vertauschen in der Grafik nicht erkennen. Es kommt zu einer Überlagerung vom linearen Vertauschen und quadratischen Vergleichen.

Das Laufzeitverhalten wird weiterhin durch die Wahl der Variablenart maßgeblich beeinflusst. Werden die Variablen als „public“ oder „private“ angelegt, so verlängert sich das Laufzeitverhalten (ca. doppelt so lang).

Hinweis: Variablen, die nicht deklariert werden, werden vom Compiler automatisch als

„private“ Variablen angelegt. Das gilt es unbedingt zu vermeiden.

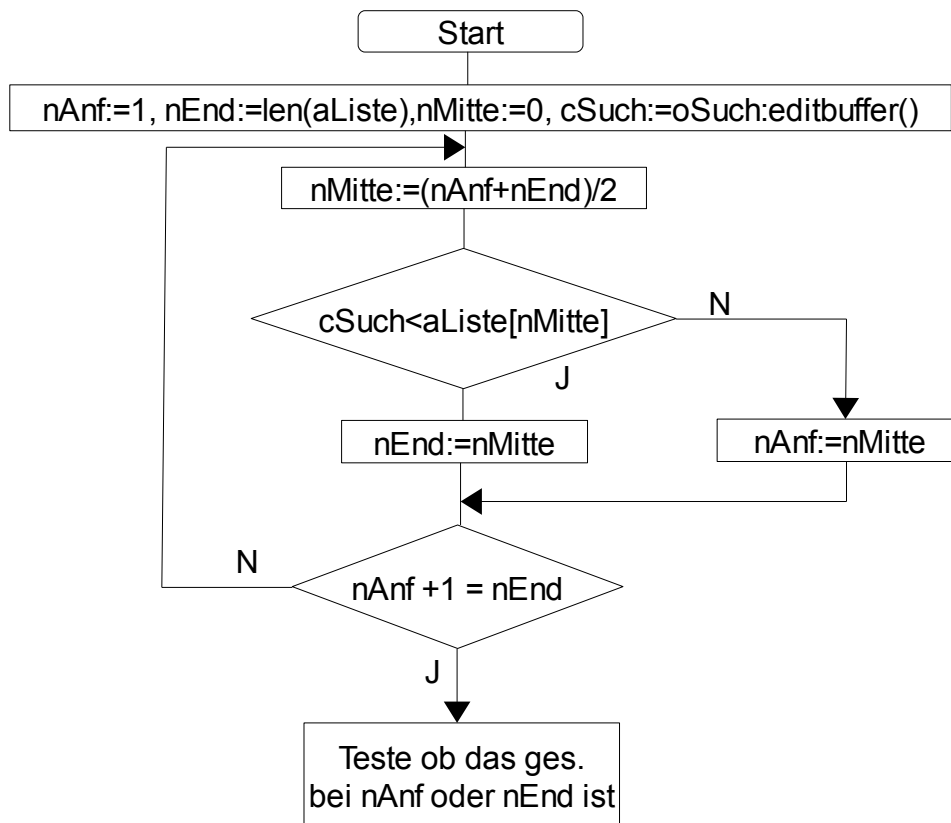
Suchen im sortierten Array

Bisher hatten wir zwei verschiedene Suchalgorithmen angesprochen.

Lineare Suche: Man beginnt beim ersten Element und sucht so lange, bis man das gewünschte Element gefunden hat. Dabei ist die Sortierung nicht von Interesse und von daher im vorliegenden Fall ohne Interesse.

Binäre Suche:

Inhalt	12	17	19	33	45	99	101	105	110	123
Pos	1	2	3	4	5	6	7	8	9	10



Vergleichen von Strings

Beim Suchen müssen Strings verglichen werden. Diese bestehen nicht nur aus verschiedenen Wörtern, sondern haben in der Regel auch eine bestimmte Länge.

Es ergeben sich eine Vielzahl von Problemen:

Im Array sind die Namen aus der Datenbank gespeichert. In der Datenbank hat jeder Name die Länge 30. Ist ein Name kleiner, so wird der Rest mit Leerzeichen aufgefüllt.

$c_{\text{Such}} = a_{\text{Liste}}[n_{\text{Anf}}]$

M	E	I	E	R	_	_	_	_	_	_	_	_	_	_
M	E	I	E	R										
ok	ok	ok	ok	ok	x									

Die 6. Position stimmt nicht mehr überein. Dort ist im oberen String ein Leerzeichen, im Unteren nichts mehr. Somit sind für ihn die Strings nicht gleich

1. Lösungsmöglichkeit: Man entfernt die Leerzeichen mit der Funktion `rtrim()` am Ende des Strings
2. Man vertauscht die beiden Variablen im Vergleich

$a_{\text{Liste}}[n_{\text{Anf}}] = c_{\text{Such}}$

15 Zeichen - 8 Zeichen. Somit werden nur 8 Zeichen beim Vergleich berücksichtigt

Damit auch Teilstrings zum Sucherfolg führen, sollten die Leerzeichen bei `cSuch` entfernt werden.